We are IntechOpen, the world's leading publisher of Open Access books Built by scientists, for scientists

4,800
Open access books available

122,000
International authors and editors

135M
Downloads

Our authors are among the

154
Countries delivered to

TOP 1%
most cited scientists

12.2%
Contributors from top 500 universities

WEB OF SCIENCE™

Selection of our books indexed in the Book Citation Index in Web of Science™ Core Collection (BKCI)

Interested in publishing with us?
Contact book.department@intechopen.com

Numbers displayed above are based on latest data collected.
For more information visit www.intechopen.com

5

Specification and Validation of Real-Time Systems Using UML Sequence Diagrams

Zbigniew Huzar and Anita Walkowiak
Wrocław University of Technology, Institute of Informatics
Poland

1. Introduction

UML (OMG, 2011) is considered as a contemporary standard in information systems development. Being a graphical modeling language it offers a family of diagrams that may be used for specification and designing of information systems. Sequence diagrams, being a part of the family, are very often used to specify functional requirements of the developed systems and are typically associated with the use case realizations in the logical view of the system under development. They show how actors involved in the scenario representing a use case realization cooperate with system's objects. Therefore, the meaning of a sequence diagram is a set of scenarios, each describing interaction between objects of the designed system and its environment. Semantics of sequence diagrams is defined informally in plain language, and, additionally, the definition is limited to the interpretation of single diagrams. But in nontrivial cases a set of sequence diagrams is necessary to give a complete specification of the system's behavior, and therefore the interpretation of the set of such diagrams is needed. Since UML has informal semantics, a set of sequence diagrams brings some interpretation problems. The problem becomes even more difficult when the real-time systems are designed when numerous time constraints are associated with the diagrams.

Hence, the primary aim of the chapter is to give a precise interpretation of a set of sequence diagrams with time constraints. The formal interpretation is necessary to construct programming tools supporting validation of the systems' behavior specification, and possibly prototyping of the systems. The chapter demonstrates how the set of scenarios specifying system's behavior may be derived from the set of sequence diagrams, and how this set may be analyzed against its consistency and completeness.

Another aim of the chapter is to propose an approach to real-time systems specification. Real-time systems have some peculiarities. For example, a typical task for a real-time system is to track the events from its environment and then responding to them, within imposed time constraints, through the generation of new events targeted to the environment. To follow such schemata, we propose to extend the UML sequence diagrams with new kinds of stereotyped combined fragments.

A specific methodological aspect of real-time system specification is also considered. Namely, very often, in addition to an explicit description of the behavior of the system, additional properties such as safety and liveness, are taken into account. Usually, the

properties are expressed in modal logics. We propose to use sequence diagrams to express them, to obtain in this way uniformity of means for a system specification. For this purpose, the notion of monitoring scenarios is introduced. Monitoring scenarios are specified by sequence diagrams, and are used to define liveness and safety properties of the system's behavior.

In the chapter, the proposed semantics of extended sequence diagrams is explained, and an example of a simple system specification and its analysis are presented. The analysis is done by means of the prototype of a programming tool that enables analysis of system's behavior against consistency and completeness as well as checking its liveness and safety properties.

The chapter is organized as follows.

Section 2 presents how UML sequence diagrams are defined, and also introduces new kinds of combined fragments that are used to define extended sequence diagrams. A set of extended sequence diagrams is used to represent the behavior of a real-time system.

Section 3 outlines our approach to specification of the real-time systems. The approach uses class diagrams to represent the structural aspect, and a set of sequence diagrams to represent the behavioral aspect of the specified system. A specific feature of the approach is a possibility to extend the behavior specification with additional monitoring diagrams – sequence diagrams – representing forbidden and expected behaviors. In this way we introduce some redundancy to the behavior specification, which enables checking safety and liveness of system's behavior. The approach is illustrated by a simple example.

In Section 4, an informal semantics of real-time system specification is explained; a notion of the graph of possible scenarios is defined. The graph is derived from the set of extended sequence diagrams, and defines a set of possible scenarios representing system's behavior.

System's specification requires validation with respect to consistency, definiteness and completeness. These properties are defined and discussed in Section 5.

Section 6 is the main section of the chapter. It formalizes semantics of a set of extended sequence diagrams. First, it defines a set of basic notions, and next it formally presents construction of the graph of possible scenarios which are semantics of a set of sequence diagrams.

Section 7 ends the chapter with concluding remarks.

2. Sequence diagrams in UML

Features of real-time systems define some requirements put on the system development process, and in consequence, on specification language used in this process. The language, in which the specification is expressed, should be characterized by the right power of expression (allowing a description of real-time systems, and development of system models for the assumed point of view), and should be abstract (allowing an appropriate level of detail description). Additionally, the language should be supported by programming tools enabling validation (confirmation that the informal user's needs are met) and verification (checking the specification against a set of properties: consistency, completeness, determinism) of specifications.

There are several languages that enable real-time systems specification, but we focus on UML, especially on UML sequence diagrams that are drawn from Message Sequence Charts (ITU-T, 2004; Cleaveland & Sengupta, 2006). UML 2.4 sequence diagrams provide mechanisms for specification of time properties (OMG, 2011).

A sequence diagram represents an interaction – a set of communications among objects arranged visually in time order. The diagram shows the objects with their lifelines participating in an interaction, and the sequences of exchanged messages, but it does not show object relationships. So, the diagram forms an interaction that consists of objects' lifelines and messages exchanged between the lifelines. For each message there are two events: sending and receiving.

The newest version of UML 2.4.1 enables explicit handling of real-time events on sequence diagrams. The basic mechanisms are: observation of current time, especially observation of time of an event occurrence, and observation of duration of message transmission. As in the previous versions of UML, time constraints may be specified – see Fig.1. The constraints may take into account times of sending and receiving a message, duration of a message transmission, times of occurring of selected events etc.

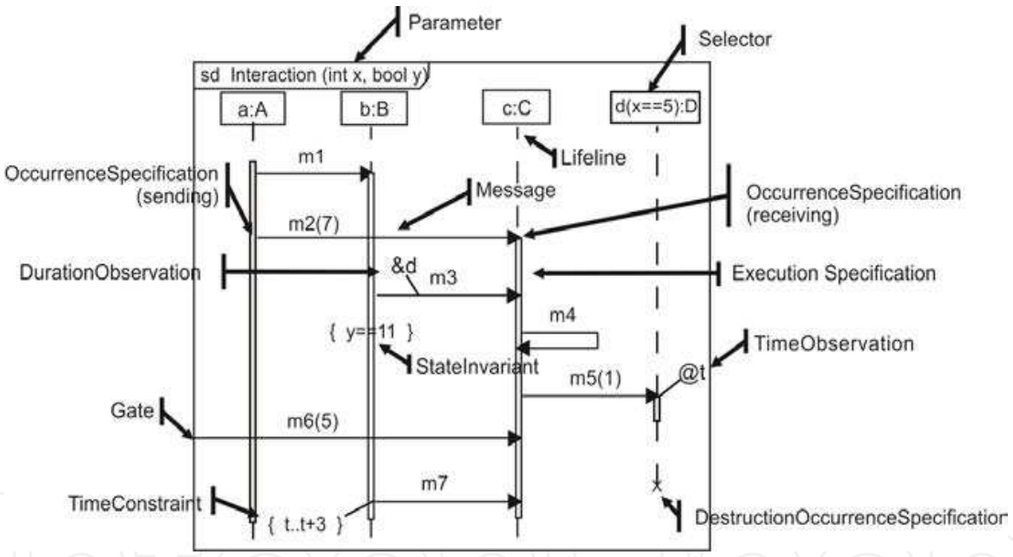

Fig. 1. Sequence diagram with time constraints

Sequence diagrams can exist in a descriptor form (describing all possible scenarios) and in an instance form (describing one actual scenario). The descriptor form uses combined fragments that are shown as nested regions within a sequence diagram. A combined fragment defines an expression of interaction fragments. A combined fragment is defined by an interaction operator and corresponding interaction operands. There are a number of combined fragments for representing contingent behavior, such as conditionals, loops, and so on. A combined fragment has a keyword, e.g., *alt*, *break*, *par*, *loop*, *seq*, *strict*, that specifies its type. Depending on the keyword, there are one or more embedded operands, each of which is a structured subfragment of the overall interaction. A combined fragment may also have zero or more gates, which specify the interface between the combined fragment and other parts of the interaction.

Following specific features of real-time systems, a new combined fragment is introduced. It is a composition of two new stereotypes «action» and «reaction» applied to arguments of the combined fragment with operator *strict* (Fig. 2a). This combined fragment expresses a typical cooperation schema between a computer system and its environment: the system should respond to signals received from the environment. The argument with the stereotype «action» specifies a message or a sequence of messages that represent a stimulus from the environment, and the argument with the stereotype «reaction» specifies a message or a sequence of messages that represent the system response. Both arguments are linked via strict sequencing operator which means that if the scenario represented by the argument «action» is executed then the scenario represented by the argument «reaction» must occur.

The other two combined fragments presented in Fig. 2b and 2c are new stereotypes of *assert* and *neg* combined fragments. The fragments are used to define liveness and safety properties, respectively, of the specified system. The first one means that if the execution reaches the beginning of the construct, then the behavior of the fragment, as an expected behavior of the specified system, must occur. The second one defines the behavior of the fragment, as a forbidden behavior of the system, may not occur.

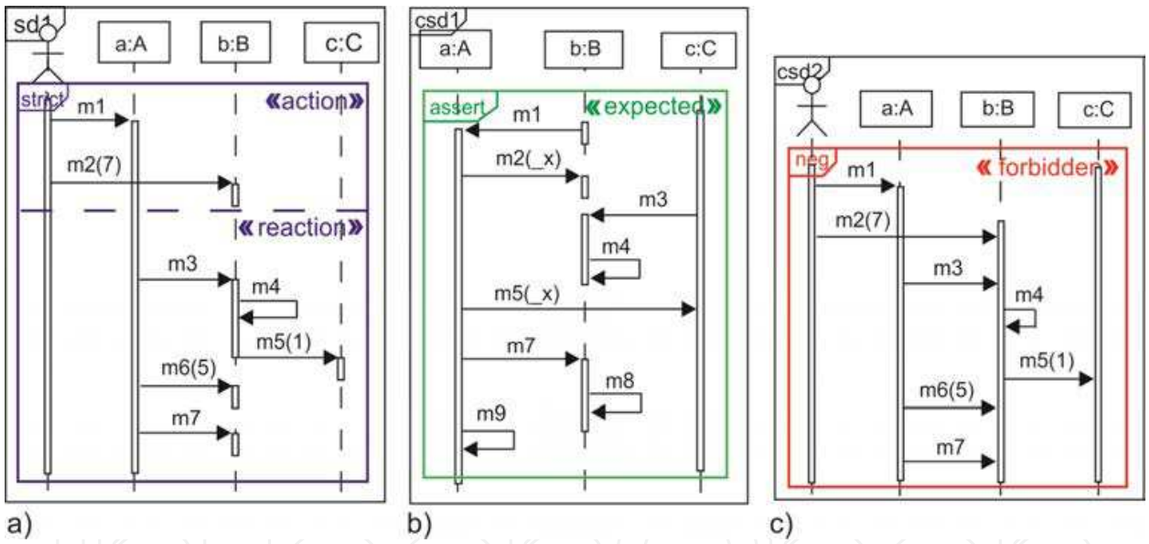

Fig. 2. Examples of sequence diagram with specific combined fragments

3. Real-time system specification

In the presented approach to system specification we define the system as a pair of two elements: the system structure that expresses the static aspect, and its behavior that presents the dynamic aspect of the system. Specification of the static aspect *sSpec* is expressed by UML class and objects diagrams while specification of the behavioral aspect *bSpec* – by a set of extended UML sequence diagrams. So, the system specification is defined as:

$$Spec_{UML} = <sSpec, bSpec>$$

In a typical approach, one would expect that the set of sequence diagrams *bSpec* will specify only the desired behavior of the developed system. Usually, such a set of sequence diagrams

reflects a family of user stories; each describing a scenario or scenarios represented by a sequence diagram.

The peculiarity of the proposed approach lies in that the set of diagrams may contain also additional diagrams called monitoring diagrams. The idea to use the monitoring diagrams comes from the postulate, that we expect high level of credibility of specification of each system, and especially specification of the real-time system. The monitoring diagrams introduce some redundancy to the specification and, in this way, increase its credibility. So, the behavioral aspect *bSpec* consists of two sets of sequence diagrams:

$$bSpec = Sd_{spec} \cup Sd_{monit}$$

where the set Sd_{spec} specifies the desired behavior of the system, and the set Sd_{monit} defines monitored behaviors. The monitored behaviors may in turn be split into forbidden and expected behaviors, namely:

$$Sd_{monit} = Sd_{forbidden} \cup Sd_{expected}.$$

The forbidden behaviors represent the safety property of the developed system, i.e., they express the fact that the undesirable situation will not appear during system execution (the system does not reach unacceptable states).

The expected behaviors represent the liveness property of the system, i.e., express the fact that if some situation is required it will happen eventually during the system execution (the system reaches desirable states).

The liveness and safety properties (Nissanke, 1997) are usually expressed in a language of modal logics (Manna & Pnueli, 1992; Manna & Pnueli, 1995). Having a model or a system's prototype, one examines whether the model or prototype complies with specified properties (*model checking*). Peculiarity of the presented approach is to use sequence diagrams to express these both properties. In this way we obtain the possibility of specifying the system's behavior and its properties using a uniform mechanism of sequence diagrams.

An example of a system specification is presented below. The example specifies a simple real-time system which controls and monitors a bakery. Fig. 3 presents a class diagram representing components of the system.

The behavior of the system is described by three user stories represented in Figs. 4, 5 and 6.

The first user story is presented on three sequence diagrams – Fig. 4. They describe reaction of the system when the main switch of the control panel is clicked to on or to off. When the main switch of the control panel is clicked to on, the main light should be turned on and the console background color should be changed to green. When the switch is clicked to off, the light should be turned off, and the console background color should be changed to white.

The second user story is presented on two sequence diagrams – Fig. 5. The story relates to an alarm situation when the current temperature of the bakery exceeds the permissible temperature for some period of time. In reaction to the situation the main light on the console is changed to red, and next, alternatively, the controller switches off the system, or the user decides about switching off or setting a new permissible temperature.

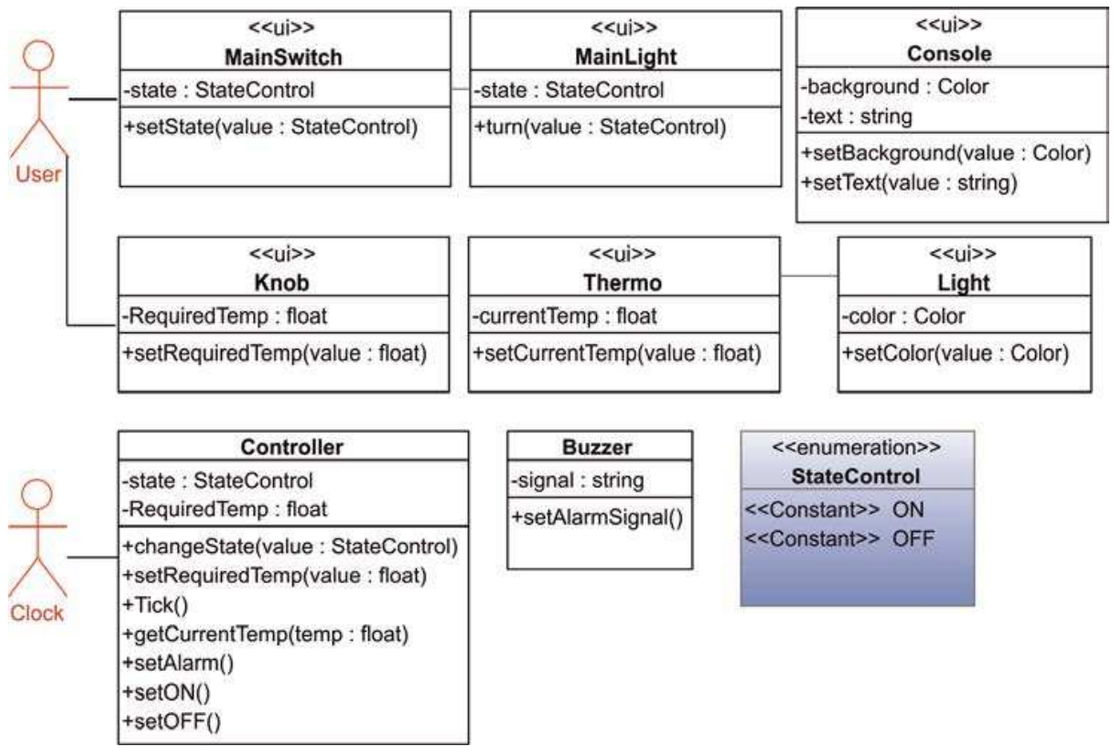

Fig. 3. Class diagram for the example

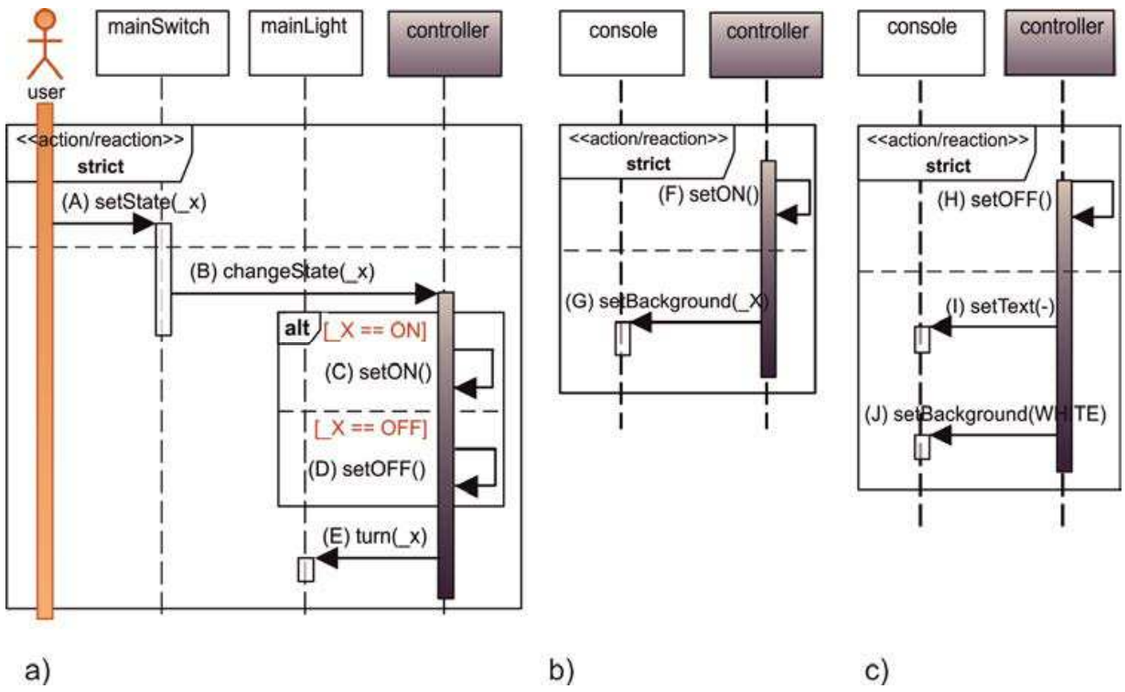

Fig. 4. Sequence diagrams – representation of the first user story

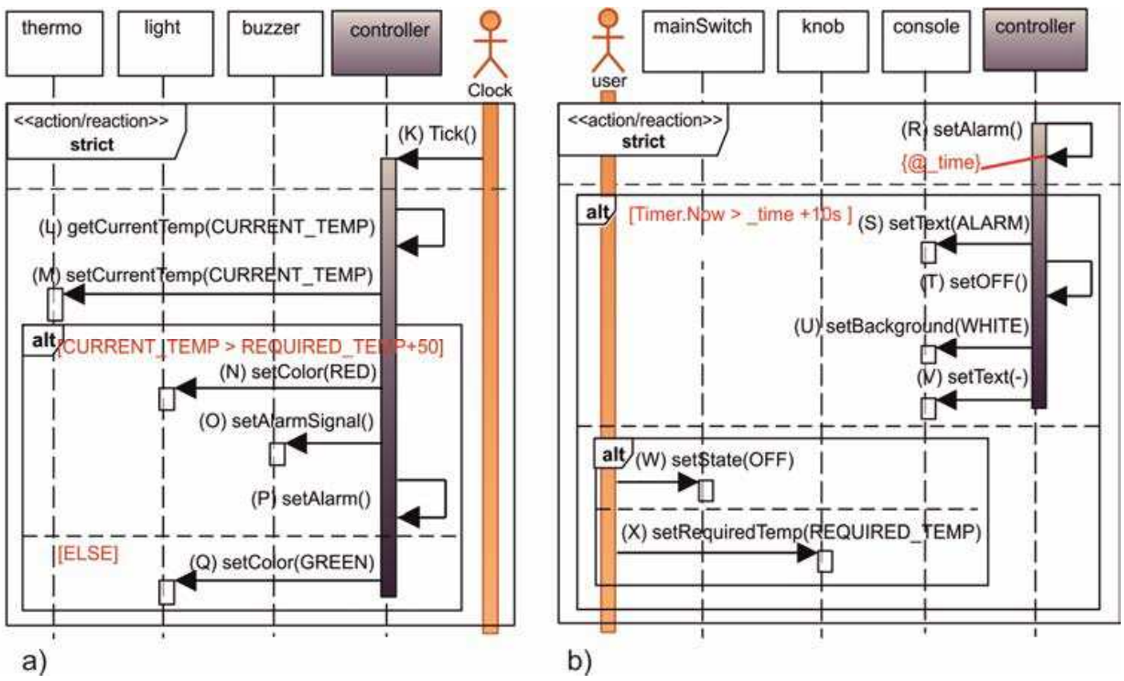

Fig. 5. Sequence diagrams – representation of the second user story

The last, a very simple user story – Fig. 6 – describes reaction of the system on setting a new permissible temperature by the user.

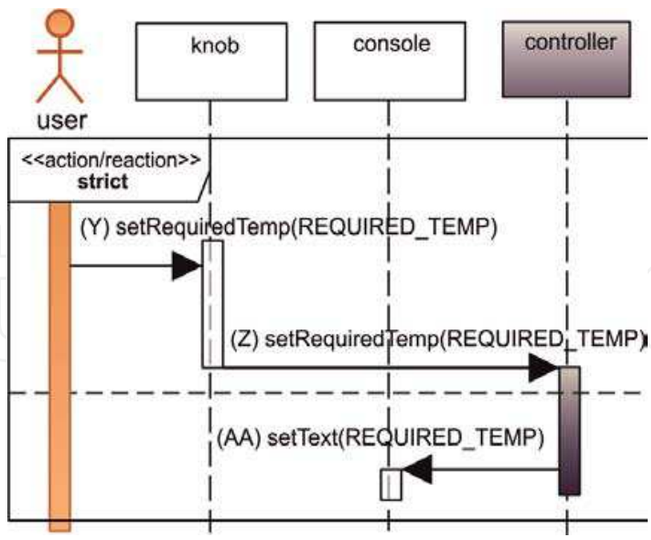

Fig. 6. Sequence diagrams – representation of the third user story

Two sequence diagrams in Fig. 7 represent the expected and forbidden behaviors of the specified system. The scenario from Fig. 7b should belong to the set of expected scenarios of the system, while the scenario from Fig. 7a should not belong to this set of scenarios.

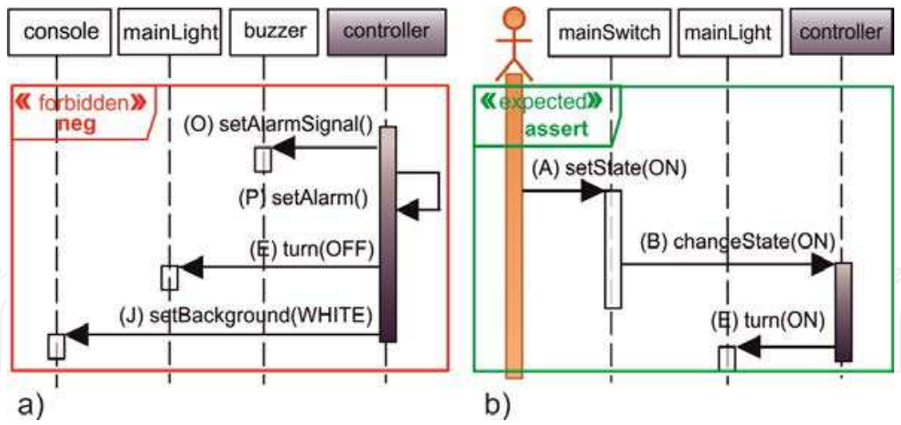

Fig. 7. Sequence diagrams – representation of expected and forbidden behaviors

4. Informal semantics of a set of sequence diagrams

The definition of UML presented in the standard (OMG, 2011) is informal. Lack of formal semantics brings ambiguity problems (Harel & Maoz, 2008; Störrle, 2004), especially in the case of automation of system development process and design of tools supporting the process. Furthermore, the possibility of UML model-checking is limited to syntax verification. To allow analysis of the properties of the spetcification formulated in UML, we propose a transformation of the specification – a set of sequence diagrams – to the abstract model which describes the behavior of the modeled system as a graph of possible scenarios (Fig. 8).

The graph of possible scenarios consists of nodes and arcs. Nodes represent system's configurations, meaning its states. Arcs represent events that cause transitions between configurations. Events may represent message sending or message receiving, time events and synchronization events. The latter are related to entering into or exiting from the combined fragments or their arguments.

The structure of the graph is similar to a tree. The root of the tree represents an initial configuration, in general, while leafs represent final configurations. It is possible however for some leafs to return to the initial configuration what is formally represented by an arc labeled with an artificial event δ. It is also possible to have loops for some configurations. The loops are labeled by synchronization events that do not change system configurations. A sequence of transitions starting from the initial configuration (a scenario) may be finite or infinite. The set of all sequences of transitions defines the set of possible scenarios of interactions between the system and its environment.

Fig. 8 presents only a fragment of the graph of possible scenarios for our exemplary specification. The graph contains only selected scenarios that are derived from the specification. The labels on arcs are symbols of messages taken from the sequence diagrams.

Now, semantics of the set of sequence diagrams may be defined as the set of all scenarios – sequences of events – generated by this graph. The scenarios are derived from fragments of interactions represented by single diagrams. On the basis of the set of sequence diagrams the graph expressing all derived scenarios is constructed.

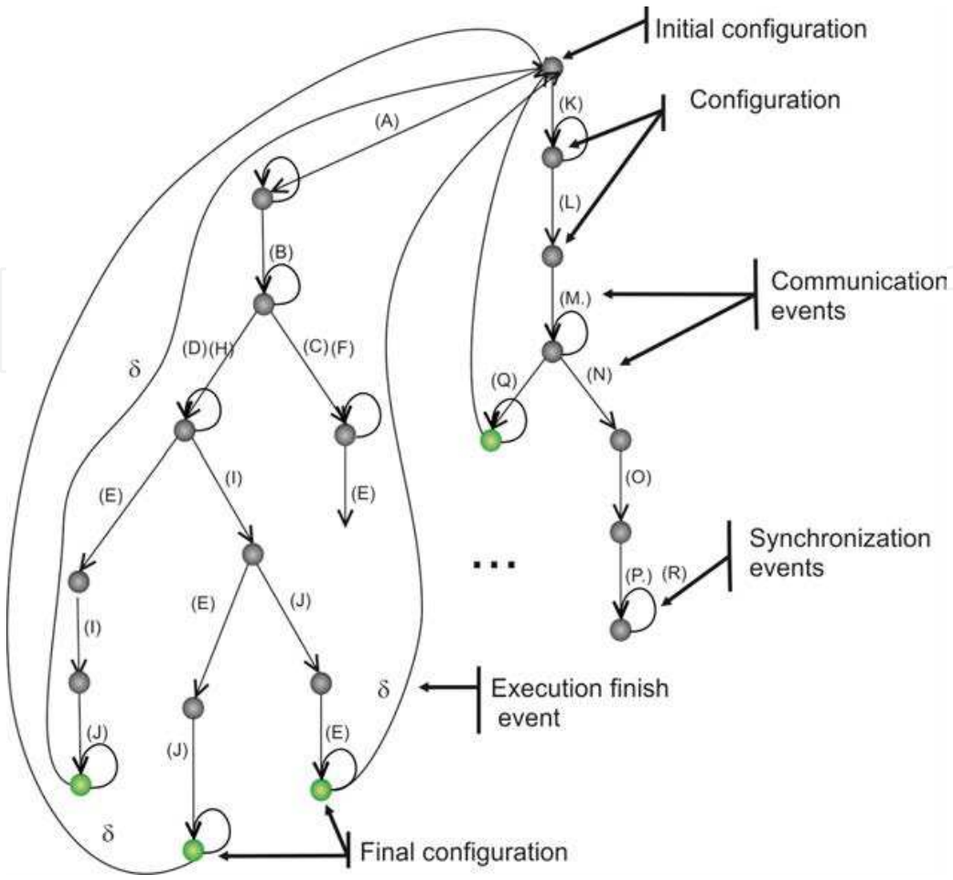

Fig. 8. Fragments of the graph of possible scenarios

The algorithm constructs the graph while walking from one location to another location along the lifelines on the sequence diagrams. A location is a time point on an object's lifeline with attached event, e.g. sending or receiving message. A set which for each lifeline of a given single diagram contains its location constitutes a snapshot of the diagram.

The snapshot represents current progress of the behavior modeled by the sequence diagram. The diagram together with its snapshot and the values of currently exchanged messages constitute a live copy of the diagram. The set of live copies of all diagrams which are involved into common scenario, i.e. operate on the same messages, determines a configuration. A configuration is changing when an event appears.

Of course, construction of the graph has to take into account all possible relationships between scenarios presented by individual sequence diagrams. In particular, in the first place, a consistency between scenarios has to be checked. The algorithm of the graph construction is presented in details in Section 6.

5. Specification properties

Each specification should be unambiguous and a complete formulation of the user's requirements. We check the specification against the following three properties: definiteness, consistency and completeness. Definiteness and consistency may be checked automatically

whereas it is not possible for completeness. Completeness refers to the domain of application and therefore it requires domain expertise; it can also be validated experimentally.

As our specifications are executable, the experiments are possible and the user observing system's behavior may decide on completeness of the specification. Nevertheless, some aspects of completeness may be checked automatically. For example, if a sequence diagram contains a message being the system response, a stimulating action is expected.

The notions of definiteness and consistency as defined below may be automatically checked. Consistency is essential and indispensable property of the specification. General definition of consistency is given in (Huzar & Walkowiak, 2010). Here, we concentrate on partial orders of exchanged messages defined by different sequence diagrams. Two orders are consistent, if the sequences determined by matching messages are the same. Consistency of specification means that partial message orders, determined by different sequence diagrams defining one scenario of interactions between the system and its environment, are consistent.

The specification is undefined if there is at least one undefined transition in a scenario derived from the set of sequence diagrams. A transition may be undefined due to an undefined value of exchanged messages.

The algorithm constructing the graph of possible scenarios is extended by the analysis of consistency, definiteness and completeness of the specification.

The analysis of the specification is carried out during the configuration transformations. Particular configurations are processed until the following appears:

- the end of interaction being a reaction to a stimulus from the environment is reached (the set of live copies of the reached configuration is empty),
- inconsistency in the message orders,
- identification of the events, which refer to indefinite values of messages (indefiniteness),
- identification of the messages exchanged between the system elements or the messages sent from the system to its environment such that they appear in activation part of a sequence diagram and there is not another sequence diagram which contains the same messages in reaction part (incompleteness).

Now we consider consistency, definiteness and completeness of the exemplary specification.

The sequence diagrams in Figs. 4c and 5b represent fragments of the same scenario of the interaction between the system and its environment. The scenario specifies the system response to the situation when the bakery temperature exceeds the desired temperature by 50^0 C by 10 seconds from the time of its detection.

Observe that there is a contradiction in ordering of the matching messages *setBackgound*(WHITE) and *setText*(-) on these diagrams. According to the diagram in Fig. 5b when the alarm is detected, the display's background is change to white and next its content is reset. According to the diagram in Fig. 4c, in this situation the controller changes its state to OFF, and the display's content is reset, and next its background is changed to white.

Using of the variable in the message C, Fig. 4b, changing display's background, results in specification indefiniteness – lack of an event, which allows defining the value of the considered variable during the execution. The variable is not symbolic (its value isn't assigned by the environment), and none of diagrams activated during the transformation

(which forms the considered interaction scenario) contains any matched message which value would be assigned to the considered variable.

The message in the activation part of the combined fragment in Fig. 6 informing about a change of the temperature requested by the user (message Z), entails incompleteness of the specification. The message specifies system's reaction to the action from its environment and therefore links two different fragments of the same scenario. However, there is no another diagram containing this message in its reaction part.

The complete graph of all possible scenarios with the results of its analysis is given in Fig. 9.

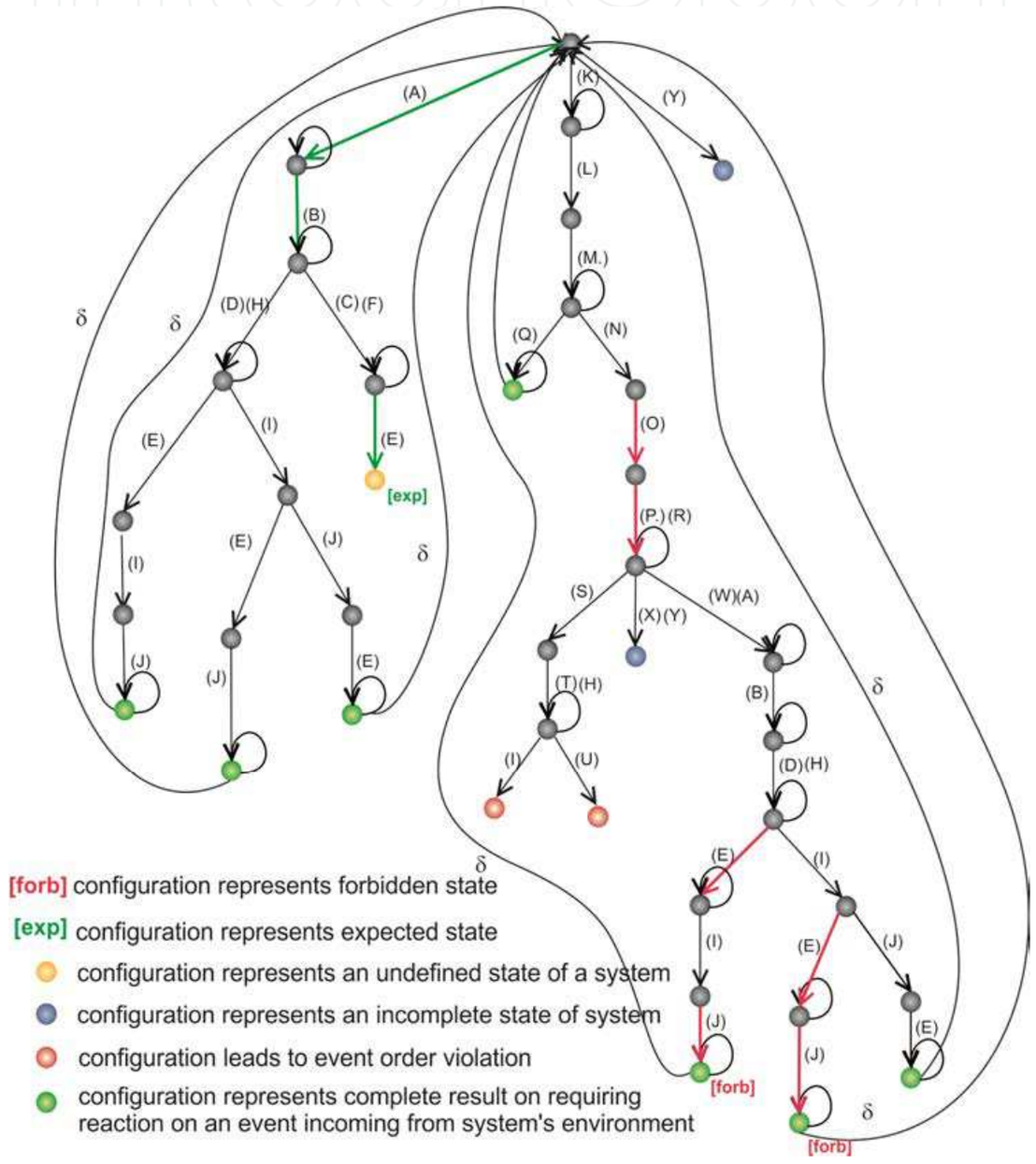

Fig. 9. Complete (interpreted) graph of possible scenarios

6. Graph of possible scenarios

The Section gives formal definitions (subsection 6.1) that are necessary to present formally construction of the graph of possible scenarios (subsection 6.2). In this way formal semantics of a set of extended sequence diagram is defined.

6.1 Basic definitions

In the definitions given below we introduce basic meta-classes from the meta-model of the extended sequence diagram. For the sake of simplicity we will treat the meta-classes as sets of specific elements. Therefore the notation $e : M$, where e is a name of an element, and M is a name of a meta-class, will be read: e is an instance of the meta-class M, or e is an element of the set M.

We will use dotted notation: if e is some element and p is its property then the expression $e.p$ means the value of the property p of the element e.

Notation and meta-elements of the system specification

$\perp$	An undefined value
$DataType$	A family of data types, each defined as a set of values and set of operations over the set of values; $d: DataType$ denotes the data type d from this family
Ω_d	A set of values of the data type $d: DataType$
Ω	A set of values of all data types: $\cup_{d:DataType}\Omega_d$
$Attribute$	A set of attributes, each defined as: $\langle name: String, d: DataType\rangle$
$Parameter$	A set of formal parameters; each defined as: $\langle name: String, d: DataType\rangle$
$Operation$	A set of operations, each defined as: $\langle name: String, parm: Parameter\rangle$
$ClassifierType$	The enumeration type of classifiers: $\{actor, external, ui, internal\}$
$Classifier$	A set of classifiers, each defined as: $\langle name: String, A: Attribute[*], O: Operation[*], type: ClassifierType\ \rangle$
$ClassifierModel$	A set of classifier models, each defined as: $\langle D: DataType[*], C: Classifier[*]\rangle$
$Instance$	A set of instances, each defined as: $\langle name: String, c: Classifier\rangle$
$InstanceModel$	A set of instances' models, each defined as: $\langle I: Instance[*]\rangle$
$StructureSpecification$	A set of system structure specifications, each defined as: $\langle cM: ClassifierModel, iM: InstanceModel\rangle$
$Environment$	A set of instances in the system's environment
$Time$	A set of time points
$LifeLine$	A set of life lines, each defined as: $\langle i: Instance, T: Time[*], t_0: Time\rangle$
$Variable$	A set of variables at a sequence diagram, each defined as: $\langle name: String, d: DataType, \omega, \Phi\rangle$, where: ω=& means symbolic (dynamically defined) variable, $\omega=\perp$ means undefined variable, and $\Phi \subseteq \Omega_d \cup I$ – defines the set of values that cannot be assigned to the variable; initially $\Phi = \emptyset$; if $\Phi \neq \emptyset$ then $\omega \in \{\perp, \&\}$.
$Message$	A set of messages, each defined as: $\langle l_{src}: LifeLine, l_{dst}: LifeLine, o: Operation, arg: Variable, sendEvent: ComEvent, recvEvent: ComEvent\rangle$
$Operator$	A set of operators of combined fragments: $\{alt, par, strict, assert, neg\}$

$CombinedFragmentArgument$	A set of arguments of a combined fragment
$CombinedFragment$	A set of combined fragments, each defined as: $\langle oper: Operator, CfL: LifeLine[*], CfA: CombinedFragmentArgument[*], guard: String\rangle$
$Event$	A set of events at a sequence diagram, each defined as: $\langle eventTime: Time[*]\rangle$
$ComEventType$	A set of types of communication events: $\{send, recv\}$
$ComEvent$	A set of communication events associated with message sending or receiving, each defined as: $\langle msg: Message, type: ComEventType\rangle$
$SyncEventType$	A set of types of synchronization events: $\{entry, exit\}$
$SyncEvent$	A set of synchronization events associated with an entry into or exit from a combined fragment or its argument, each defined as: $\langle f: CombinedFragment \cup CombinedFragmentArgument, type: SyncEventType\rangle$
$SequenceDiagram$	A set of sequence diagrams, each defined as: $\langle name: String, L: LifeLine[*], Var: Variable[*], Msg: Message[*], Cf: CombinedFragment[*], cf_0: CombinedFragment, E_{com}: ComEvent[*], E_{sync}: SyncEvent[*]\rangle$
$BehaviourSpecification$	A set of system behavior specifications, each defined as: $\langle Sd_{spec}: SequenceDiagram[*], Sd_{monit}: SequenceDiagram[*] = Sd_{forbidden} \cup Sd_{expected}\rangle$
src	The function $src: Message \longrightarrow Instance$ yields the instance which sends a message: $src(msg) = msg.l_{src}.i$
dst	The function $dst: Message \longrightarrow Instance$ yields the instance which receives a message: $dst(msg) = msg.l_{dst}.i$
$action$	The function $action: SequenceDiagram \longrightarrow CombinedFragmentArgument$ for a given sequence diagram, yields the argument of a combined fragment stereotyped by «$action$»
$reaction$	The function $reaction: SequenceDiagram \longrightarrow CombinedFragmentArgument$ for a given sequence diagram, yields the argument of a combined fragment stereotyped by «$reaction$»
$enclosing$	The function $enclosing: Message \cup Event \cup CombinedFragment \longrightarrow CombinedFragmentArgument \cup \{\perp\}$ for a given specification element, yields an argument of the combined fragment in which the element is nested directly; if there is no such an argument the function returns $\perp$
$allEnclosing$	The function $allEnclosing: Message \cup Event \cup CombinedFragment \longrightarrow 2^{CombinedFragmentArgument}$ for a given specification element, yields a list of arguments of all combined fragment in which the element is nested:
$\preccurlyeq_{sd} \subseteq Time^2$	A partial ordering relations defined on the set of time points at lifelines of a given extended sequence diagram
$\preccurlyeq_{sd} \subseteq Event^2$	A partial ordering relations defined on the set of events of a given extended sequence diagram

The series of the following definitions refines some notions, e.g. snapshot, configuration, which were informally introduced in Section 4.

Definition 6.1.1

A snapshot of a sequence diagram sd, noted $s_{sd}: Snapshot$, is defined as the set:

$$s_{sd} = \{\langle l, t\rangle | \; l \in sd.L, t \in l.T\} \tag{6.1.1}$$

where:

- l – is a lifeline of an object from the diagram *sd*, and
- t – is a time instance at the lifeline l representing time of a communication event or a synchronization event on entry to or exit from a combined fragment.

The function $initial: SequenceDiagram \longrightarrow Snapshot$ yields an initial snapshot of a sequence diagram.

An example of a snapshot of a sequence diagram is presented in Fig. 10.

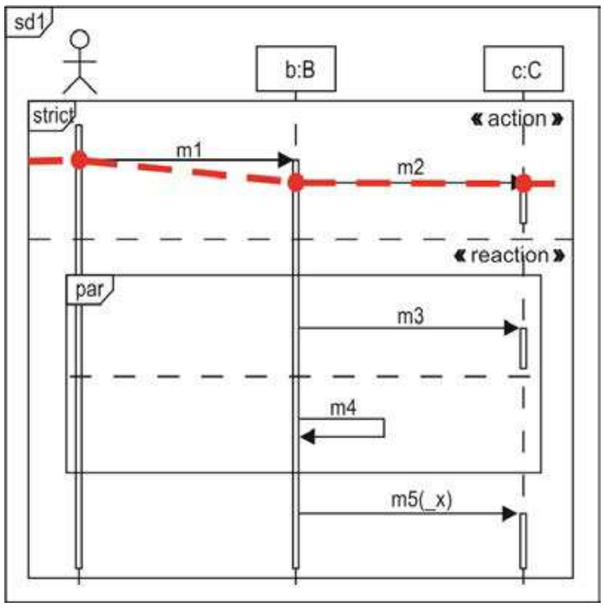

Fig. 10. An example of a snapshot

Let $l.t_i$ means the *i*-th time point at the life line l of the sequence diagram.

Definition 6.1.2

A set of time points $T \subseteq \bigcup_{l \in sd.L} l.T$ is said to be closed when:

$$\forall_{l \in sd.L}: \forall_{l.t_x \in T}: y \leq x \iff l.t_y \in T \tag{6.1.2}$$

Definition 6.1.3

A lower closure $\downarrow s_{sd}$ of a snapshot s_{sd} is defined as the minimal closed set of time points of the diagram *sd* which contains all points of the snapshot.

A snapshot s_{sd} is correct if:

$$\forall_{\langle l,t\rangle \in s_{sd}}: \forall_{t' \in l.T}: t' \preccurlyeq_{sd} t \iff t' \in \downarrow s_{sd} \tag{6.1.3}$$

Definition 6.1.4

An active copy of the sequence diagram *sd*, noted lc_{sd}: $LiveCopy$, is defined as the triple:

$$lc_{sd} = \langle sd, s_{sd}, mode \rangle \tag{6.1.4}$$

where:

- sd – is a copy of the sequence diagram with the valuation of its variables,
- s_{sd} – a snapshot of the sequence diagram showing the progress of the behavior described by the diagram, and
- $mode \in \{action, reaction, check\}$ – a state of the active copy. The active copy is in the $action$ state if realizes behavior specified in the activation part of the diagram, in the $reaction$ state if realizes behavior specified in the reactive part of the sequence diagram, and in the $check$ state if realizes behavior specified by the monitoring diagram.

Let an active copy and a variable with the valuation established for the active copy be given (see example in Fig. 11). Then, the function val: $LiveCopy \times Variable \longrightarrow \Omega \cup \{\perp, \&\}$ yields a value of a given variable, the function $forbiddenVal$: $LiveCopy \times Variable \longrightarrow 2^{\Omega \cup \{\perp, \&\}}$, yields a set of forbidden values for a given variable, and the function $free$: $LiveCopy \times Variable \longrightarrow \{true, false\}$ indicates by $true$ or *false* whether valuation of the variable is undefined.

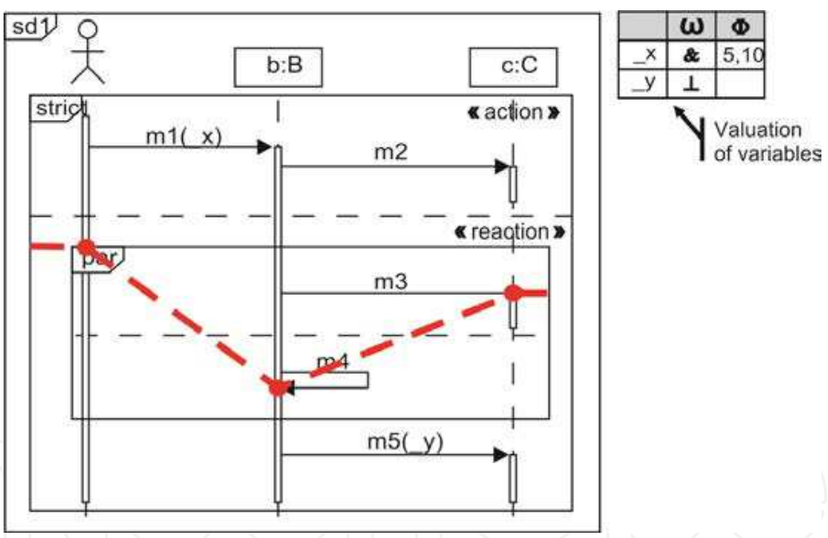

Fig. 11. An example of an active copy

If an active copy is known and a given event happens then by applying the function $advance$: $LiveCopy \times Event \longrightarrow LiveCopy$ a new active copy is defined.

Definition 6.1.5

A configuration of a set of sequence diagrams, noted $conf$: *Configuration*, is defined as the tuple:

$$conf = \langle e, Lc_{spec}, Lc_{forbidden}, Lc_{expected}, context, time_{context}, guards, Forbidden, Expected, type \rangle \tag{6.1.5}$$

where:

- e – is a communication event which causes a transition to a new state represented by the configuration, $e = \perp$ for the initial configuration,
- Lc_{spec} – a subset of active copies of the sequence diagrams specifying required behavior,
- $Lc_{forbidden}$ – a subset of active copies of the sequence diagrams specifying forbidden behavior,
- $Lc_{expected}$ – a subset of active copies of the sequence diagrams specifying expected behavior,
- $context$ – a set of bound variables, which values are agreed during a transition to the configuration,
- $time\_context$ – a set of time instants of the event e and other events associated with e,
- $guards$ – a condition for realization of the communication event e,
- $Forbidden$ – a set of sequence diagrams specifying forbidden behavior,
- $Expected$ – a set of sequence diagrams specifying expected behavior,
- $type \in \{complete, violating, undefined, incomplete, \perp\}$ – type of the final configuration; $complete$ – indicates that the configuration is the result of a completely executed reaction to the event incoming from system's environment; $violating$ – indicates whether the configuration leads to the violation of event order; $undefined$ – indicates whether the configuration represents an undefined state of a system; $incomplete$ – indicates whether the configuration represents an incomplete state of system; $\perp$ indicates the leaf-configuration.

An example of a configuration of a set of sequence diagrams is presented in Fig. 12.

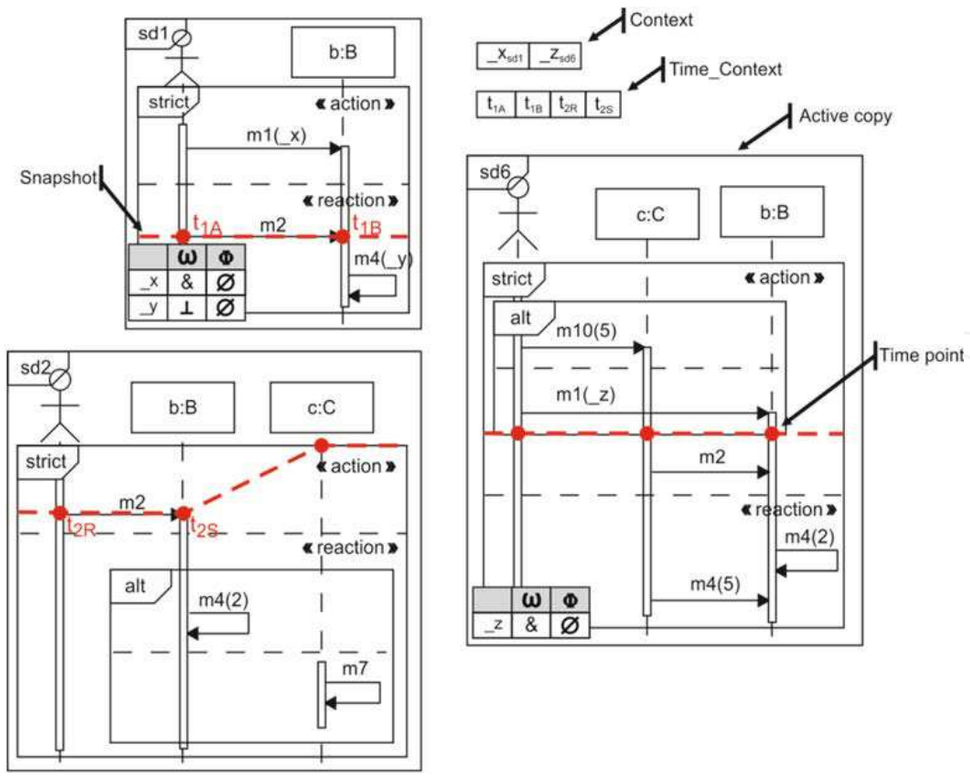

Fig. 12. An example of a configuration

The function $areConnect: Configuration \times Variable \times Variable \longrightarrow \{true, false\}$ returns $true$ if the variables are bound, and $false$ otherwise.

Definition 6.1.6

An event *e* is a **first event** in the diagram *sd* if it belongs to the set of communication events of the diagram, and it is a message sending event, and there are no other events that precede *e* in the sense of the relation $\preccurlyeq_{sd}$:

$$e \in sd.E_{first} \iff \left(e \in sd.E_{com} \wedge e.type = send \wedge \nexists_{e' \in sd.E_{com}} : e' \preccurlyeq_{sd} e\right) \quad (6.1.6)$$

Examples of first events for sequence diagrams are presented in Fig. 13.

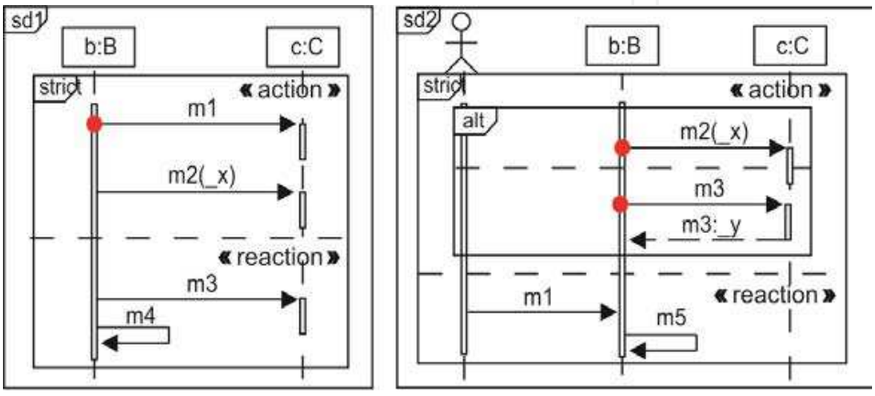

Fig. 13. Examples of first events for the sequence diagrams

Definition 6.1.7

An event *e* is an **initial event** of the sequence diagram *sd* if it belongs to the set of first events of the diagram and it is message event which is sent by the environment of the system:

$$e \in sd.E_{init} \iff \left(e \in sd.E_{first} \wedge src(e.msg) \in Environment\right) \quad (6.1.7)$$

An example of an initial event for a sequence diagram is presented in Fig. 14.

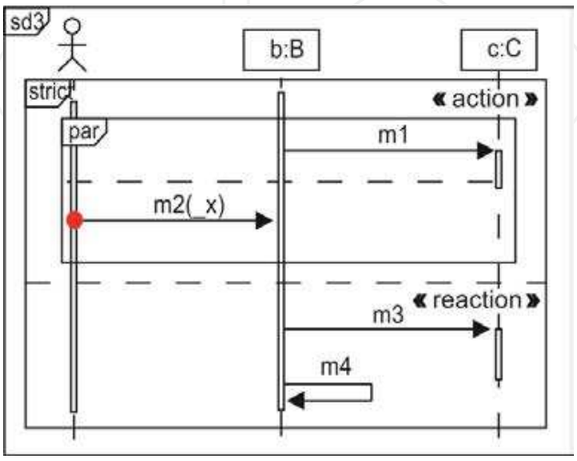

Fig. 14. An example of the initial event for the sequence diagram

Definition 6.1.8

An event *e* is **enabled** in the active copy lc if it belongs to the set of events of the diagram $lc.sd$, and the time points of the snapshot for all lifelines representing instances participating in realization of the event *e* directly precede *e*, and there is no a such event *e'* that proceeds *e* (does not belong to the lower closure of the snapshot $\downarrow lc.s_{sd}$).

If *e* is a communication event sending a message and *e''* is a communication event receiving the message then *e''* is enabled in the snapshot representing realization of the event *e*:

$$e \in lc.E_{enabled} \Leftrightarrow \tag{6.1.8}$$

$$\left(e \in lc.sd.E_{com} \cup lc.sd.E_{sync} \wedge \nexists_{e' \in lc.sd.E_{com} \cup lc.sd.E_{sync}} : (e' \prec_{sd} e \wedge e' \notin \downarrow lc.s_{sd}) \wedge \right.$$
$$\left.(e.type = send \Longrightarrow e.msg.recvEvent \in advance(lc.s_{sd}, e).E_{enabled})\right)$$

An example of time point associated with an enabled event for an active copy of a sequence diagram is presented in Fig. 15.

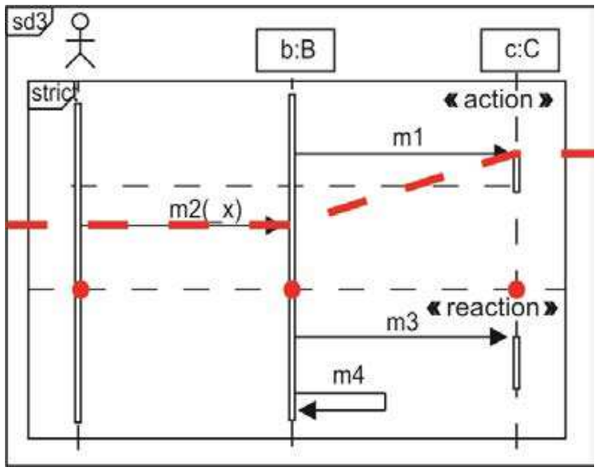

Fig. 15. An example of time point associated with an enabled event for an active copy

Definition 6.1.9

An event *e* **causes a transition** of the active copy lc if:

- it belongs to the set of enabled events for the active copy, and
- if it is associated with the message sending by the system then it also belongs to the scenario representing system's reaction (part «$reaction$») and the variable to which it refers has defined value:

$$e \in lc.E_{advanced} \Leftrightarrow \tag{6.1.9}$$

$$e \in lc.E_{enabled} \wedge$$
$$\left(e \in lc.sd.E_{com} \wedge src(e.msg) \notin Environment \Longrightarrow \left(reaction(lc.sd) \in allEnclosing(e) \wedge \right.\right.$$
$$\left.\left.(e.msg.arg \neq \perp \Longrightarrow val(lc, e.msg.arg.\omega) \neq \perp)\right)\right)$$

An example of time point associated with an event causing transition for an active copy of a sequence diagram is presented in Fig. 16.

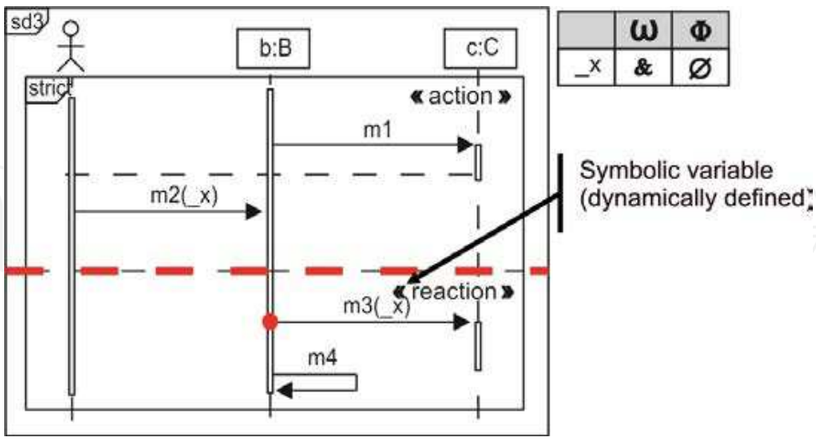

Fig. 16. An example of a time point associated with the event causing transition for an exemplary active copy

Definition 6.1.10

An event e is **reachable** for the active copy lc if it is the enabled communication event for the copy or if there exists a synchronization event e' such that e is reachable for the active copy representing realization of the event e':

$$e \in lc.E_{reachable} \iff \tag{6.1.10}$$

$$\left(e \in lc.sd.E_{com} \wedge \left(e \in lc.E_{enabled} \vee \exists_{e' \in lc.sd.E_{sync}} : e \in advance(lc, e').E_{reachable}\right)\right)$$

An example of a time point associated with the reachable event for an active copy is presented in Fig. 17.

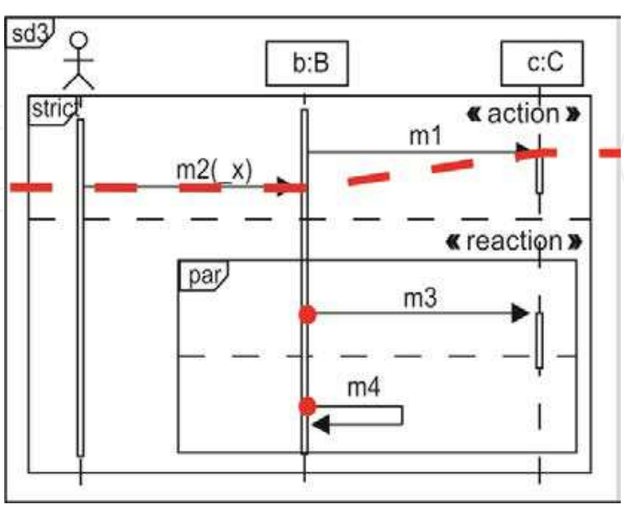

Fig. 17. An example of a time point associated with the reachable event for an exemplary active copy

Definition 6.1.11

An event e **violates events' ordering** of the active copy lc if it belongs to the set of obligatory communication events of the diagram $lc.sd$, but it is not reachable for the copy:

$$e \in lc.E_{violating} \Leftrightarrow \tag{6.1.11}$$

$$\left(e \in lc.sd.E_{com} \wedge e \notin lc.E_{reachable} \wedge \nexists_{cfa \in lc.sd.Cf_{alt}.CfA} : cfa \in allEnclosing(e)\right)$$

An example of time points associated with violating events for an exemplary active copy is presented in Fig. 18.

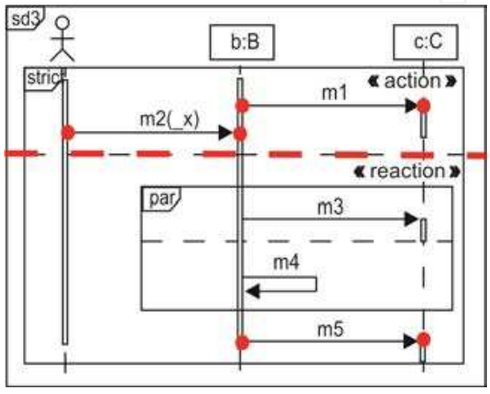

Fig. 18. An example of time points associated with the violating events for an exemplary active copy

Definition 6.1.12

A **variable** var' is ***weak-unifiable*** with a variable var in a given configuration if:

- its value is not defined,
- has the same value as the variable var, for symbolic variable the variables are bound,
- the value of the variable var' is defined dynamically, the value of the variable var is defined statically and does not belong to the set of forbidden values of the variable var':

$$match_{w-unif}(var', var) \stackrel{\text{def}}{=} (var'.\omega = \perp \vee \left(var'.\omega = var.\omega \wedge \left(var'.\omega = \& \Longrightarrow areConnect(var', var)\right)\right) \vee (var'.\omega = \& \wedge var.\omega \notin \{\perp, \&\} \wedge var.\omega \notin var'.\Phi)) \tag{6.1.12}$$

Definition 6.1.13

A **variable** var' is ***strict-unifiable*** with a variable var if has the same value, for symbolic variable the variables are bound:

$$match_{s-unif}(var', var) \stackrel{\text{def}}{=}$$

$$\left(var'.\omega = var.\omega \wedge \left(var'.\omega = \& \Longrightarrow areConnect(var', var)\right)\right) \tag{6.1.13}$$

$var'.\omega$	$var.\omega$	$match_{w-unif}(var', var)$	$match_{s-unif}(var', var)$
x	x	$true$	$true$
&	x	$true$, if $x \notin var'.\Phi$	$false$
$\perp$	x	$true$	$false$
x	&	$false$	$false$
&	&′	$true$, if $areConnect(var', var)$	$true$, if $areConnect(var', var)$
$\perp$	&	$true$	$false$
x	$\perp$	$false$	$false$
&	$\perp$	$false$	$false$
$\perp$	$\perp$	$true$	$true$

Table 1. Definition of the strict-unifiable and weak-unifiable variables, where x is a concrete value.

Definition 6.1.14

A **message** msg' is ***weak-unifiable*** (***strict-unifiable***) with a message msg when both messages are sent between the same objects, and relate to a call of the same operation and have *weak-unifiable* (*strict-unifiable*) arguments:

$$match_{q-unif}(msg', msg) \stackrel{\text{def}}{=} (src(msg') = src(msg) \wedge dst(msg') = dst(msg) \wedge msg'.o = msg.o \wedge match_{q-unif}(msg'.arg, msg.arg)) \quad (6.1.14)$$

where q means '*weak*' or '*strict*'.

Definition 6.1.15

A **communication event** e' is ***weak-unifiable*** (***strict-unifiable***) with a communication event e when both events are of the same type (sending or receiving), and relate to a *weak-unifiable* (*strict-unifiable*) message:

$$match_{q-unif}(e', e) \stackrel{\text{def}}{=} \left(e'.type = e.type \wedge match_{q-unif}(e'.msg, e.msg)\right) \quad (6.1.15)$$

where q means '*weak*' or '*strict*'.

Let $bSpec$: $BehaviourSpecification$ means the specification of the system behavior expressed by a set of extended UML sequence diagrams.

Definition 6.1.16

A graph of possible scenarios representing system's interaction with its environment is defined as:

$$G_{bSpec} = \langle V_{bSpec}, A_{bSpec}, guard \rangle \quad (6.1.16)$$

where:

- V_{bSpec} – a set of graph vertices; each vertex is labeled by a configuration which is reachable from the initial configuration,
- A_{bSpec} – a set of graph arcs; each arc is labeled by an event; each arc has the form $\langle u, v, e\rangle \in A_{bSpec}$, where $u, v \in V_{bSpec}$ are vertices labeled by $c_1, c_2 : Configuration$, and $e : Event$ is the event which causes a transition from c_1 to c_2, noted by $c_1 \overset{e}{\rightarrow} c_2$,
- $guard : A_{bSpec} \longrightarrow Exp \cup \{\perp\}$ – a partial function that assigns a Boolean expression to an arc.

Further, we will use the function, $conf : V_{bSpec} \longrightarrow Configuration$, which for a given vertex of the graph of possible scenarios returns a configuration labeling the vertex.

6.2 Construction of the graph of possible scenarios

The algorithm of construction of the graph of possible scenarios is presented below. To facilitate presentation of the algorithm a state machine diagram presenting an active copy – a component of a transformed configuration – is shown in Fig. 19. A detailed description of the algorithm is summarized in the form of an activity diagram in Fig. 20, after its textual description.

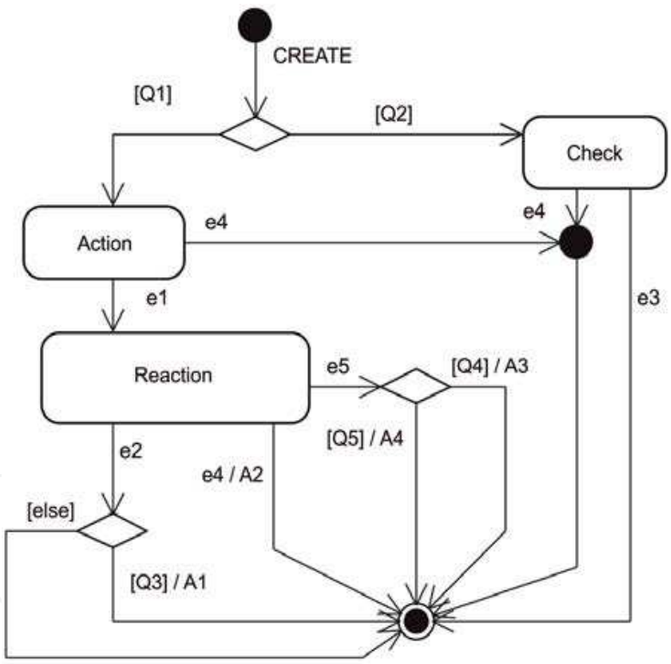

Fig. 19. State machine for an active copy, where:

TRIGGERS:

CREATE - One of the first events of the diagram occurs

- e1 – The SyncEvent - exit from the argument with the stereotype «action» of the diagram occurs
- e2 – The SyncEvent - exit from a combined fragment with the stereotype «action»/«reaction» of the diagram occurs

- e3 – The SyncEvent - exit from a combined fragment with the stereotype «forbidden» or «expected» of the diagram occurs
- e4 – One of the violates events' ordering events of the diagram occurs
- e5 - when(condition), where the condition is defined as: 'The set of events causing transitions for active copies within the configuration is empty'.

GUARDS:

- Q1 - The specification diagram
- Q2 - The monitoring diagram
- Q3 - Active Copy is the last element of the set of active copies of the configuration
- Q4 - The set of events that are accessible at the given configuration contains the event which refers to a variable whose value is not specified
- Q5 - The set of events that are accessible at the given configuration contains the system event which belongs to the scenario representing system activation

ACTIVITIES:

- A1 - Label the configuration as complete final configuration
- A2 - Label the configuration as inconsistent final configuration
- A3 - Label the configuration as unspecified final configuration which represents system's deadlock
- A4 - Label the configuration as incomplete final configuration which represents system's deadlock

The vertices and arcs of graph $G_{bSpec} = \langle V_{bSpec}, A_{bSpec}, guard \rangle$ are constructed iteratively starting from:

- $V_{bSpec} = \{v_0\}$, where v_0 is labeled by the initial configuration: $c_0 = \langle \perp, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \perp \rangle$,
- $A_{bSpec} = \emptyset$,
- $guard: A_{bSpec} \longrightarrow Exp \cup \{\perp\}$ – a partial function that assigns a Boolean expression to an arc.

1. The set of events that initiate system's behavior is defined:

$$E_{init} = \left\{ e | e \in \cup_{sd \in bSpec.Sd_{spec}} sd.E_{init} \right\}.$$

2. The set E_{init} is factorized (elimination of repetitions, marking symbolic variables, setting forbidden values for symbolic variables):

$$factore(c_0, E_{init}) \qquad /* \text{ definition 6.2.1}$$

3. For each initiate event $e \in E_{init}$:
   a. The transition is computed $c_0 \xrightarrow{e}_{com} c$, /* definition 6.2.3
   b. let v be a vertex labeled by the configuration c:

$$V_{bSpec} \leftarrow V_{bSpec} \cup \{v\},$$

   c. let a be an arc of the form $\langle v_0, v, e \rangle$:

$$A_{bSpec} \leftarrow A_{bSpec} \cup \{a\}.$$

4. For each leaf-vertex $v \in V_{bSpec}$, which is not labeled by a final configuration c such that $c = conf(v)$ $(c.type = \perp)$:
   a. If the set of active copies of the configuration is empty $(c.Lc_{spec} = \emptyset)$ return to the initial configuration in which the system awaits for events from its environment:
      i. the configuration is considered as complete final configuration:

$$c.type \leftarrow complete,$$

      ii. let a be an arc of the form $\langle v, v_0, \delta \rangle$:

$$A_{bSpec} \leftarrow A_{bSpec} \cup \{a\},$$

      iii. go to step 4.
   b. The set of events which enable transitions for active copies within the configuration c is defined:

$$E_{advanced} = \left\{ e | e \in \bigcup_{lc \in c.Lc_{spec}} lc.E_{advanced} \right\}.$$

   c. If the set $E_{advanced}$ contains synchronization events, i.e.

$$\left(\exists_{e \in E_{advanced}} : e \in \bigcup_{sd \in bSpec} sd.E_{sync} \right):$$

      i. then for each synchronization event:
      a. define transition $c \xrightarrow{e}_{sync} c'$,
      b. let a be an arc of the form $\langle v, v, e \rangle$: /* definition 6.2.2

$$A_{bSpec} \leftarrow A_{bSpec} \cup \{a\},$$

      ii. go to step 4.
   d. If the set $E_{advanced}$ contains communication events, i.e.

$$\left(\exists_{e \in E_{advanced}} : e \in \bigcup_{sd \in bSpec} sd.E_{com} \right):$$

      i. factorization of the set (elimination of copies, marking the symbolic variables, determining forbidden values of the symbolic variables):

$$factore(c, E_{advanced}).$$ /* definition 6.2.1

      ii. for each communication event:

      a. the transition is calculated $c \xrightarrow{e}_{com} c'$, /* definition 6.2.3
      b. let v' be a vertex labeled by the configuration c':

$$V_{bSpec} \leftarrow V_{bSpec} \cup \{v'\},$$

      c. let a be an arc of the form $\langle v, v', e \rangle$:

$$A_{bSpec} \leftarrow A_{bSpec} \cup \{a\}.$$

      iii. go to step 4.

e. If the set is empty ($E_{advanced} = \emptyset$) then:
   i. If the set of events that are accessible at the given configuration contains the event which refers to a variable whose value is not specified, i.e.

$$\left(\exists_{e \in \bigcup_{lc \in c.Lc_{spec}} lc.E_{enabled}}: \left(e \in \bigcup_{sd \in bSpec} sd.E_{com} \wedge e.type = send \wedge e.msg.arg \neq \perp \wedge e.msg.arg.\omega = \perp\right)\right)$$

   the configuration is considered as unspecified final configuration which represents system's deadlock:

$$c.type \leftarrow undefined,$$

   ii. If the set of events that are accessible at the given configuration contains the system event which belongs to the scenario representing system activation, i.e.

$$\left(\exists_{e \in \bigcup_{lc \in c.Lc_{spec}} lc.E_{enabled}}: \left(e \in \bigcup_{sd \in bSpec} sd.E_{com} \wedge src(e.msg) \notin Environment \wedge action(sd) \in allEnclosing(e)\right)\right)$$

   the configuration is considered as incomplete final configuration which represents system's deadlock:

$$c.type \leftarrow incomplete,$$

   iii. go to step 4.

Definition 6.2.1

Factorization of the set of the activating events which cause transitions (elimination of copies, marking the symbolic variables, determining forbidden values of the symbolic variables) – $factore(conf: Configuration, E: ComEvent\ [*])$ is defined as follows:

Let the set of events with a fixed valuation of variables is empty, i.e. $E_{out} = \emptyset$:

1. Create a copy of the events:

$$\forall_{e \in E}: E_{out} \leftarrow E_{out} \cup \{\ copy(e)\ \}.$$

2. Remove repetition of the *strict-unifiable* events:

$$\forall_{e,e' \in E_{out}} : e \neq e' \wedge conf.match_{s-unif}(e, e') \Longrightarrow E_{out} \leftarrow E_{out} \setminus \{e'\}.$$

3. For each event $e \in E_{out}$, which has a statically undefined variable, extend the set of forbidden values about the statically defined values of variables of the events $e' \in E_{out}$, which are *weak-unifiable* with *e*:

$$\forall_{e,e' \in E_{out}} : e \neq e' \wedge conf.match_{w-unif}(e, e') \wedge var \neq \perp \wedge var.\omega \in \{\perp, \&\} \wedge var'.\omega \notin \{\perp, \&\} \Longrightarrow var.\Phi \leftarrow var.\Phi \cup \{var'.\omega\}$$

where:

$$var = e.msg.arg,$$

$$var' = e'.msg.arg.$$

4. Each event sending by system's environment, which refers to a variable with statically undefined value, declare/label its variable as symbolic:

$$\forall e \in E_{out}: e.msg.arg \neq \perp \wedge e.msg.arg.\omega = \perp \wedge src(e.msg) \in Environment \Longrightarrow e.msg.arg.\omega \leftarrow \&$$

5. $E_{in} \leftarrow E_{out}$

Definition 6.2.2

The transition relations $c \xrightarrow{e}_{sync} c'$ is defined by the following rules:

For each active copy of the configuration c $\left(lc \in c.Lc_{spec} \cup c.Lc_{forbidden} \cup c.Lc_{expected}\right)$ for which the event e is reachable ($e \in lc.E_{reachable}$):

1. modify the snapshot of the active copy:

$$lc \longleftarrow advance(lc, e).$$

2. if the event *e* represents an entry to a combined fragment ($e = \langle cf, entry \rangle$):

   a. which is a strict sequence, negation or assertion combined fragment ($cf.oper \in \{strict, assert, neg\}$) then:

$$lc \longleftarrow advance(lc, \langle cf.cfa_1, entry \rangle),$$

   b. which is a parallel or alternative combined fragment ($cf.oper \in \{par, alt\}$) then:

$$\forall_{cfa \in cf.CfA}: lc \longleftarrow advance(lc, \langle cfa, entry \rangle).$$

3. if the event *e* represents an exit from an argument of a combined fragment ($e = \langle cf.cfa_i, exit \rangle$):

   a. which is a strict sequence ($cf.opere = strict$):

      i. if there exists a subsequent argument of the combined fragment then entry to the argument:

$$lc \longleftarrow advance(lc, \langle cf.cfa_{i+1}, entry \rangle),$$

      ii. if the event *e* is an exit from the argument with the stereotype «action» $\left(cf.cfa_i = action(lc.sd)\right)$ then:

$$lc.mode \longleftarrow reaction,$$

   b. which is a parallel ($cf.opere = par$), and all other arguments of this combined fragment reached their end $\left(\forall_{cfa_j \in cf.CfA, j \neq i} \langle cfa_j, exit \rangle \in \downarrow lc.s_{sd}\right)$, then exit the entire combined fragment:

$$lc \longleftarrow advance(lc, \langle cf, exit \rangle),$$

c. which is alternative ($cf.opere = alt$) then exit the combined fragment:

$$lc \longleftarrow advance(lc, \langle cf, exit \rangle).$$

4. if the event *e* represents an exit from a combined fragment ($e = \langle cf, exit \rangle$):

   a. which is a strict sequence with «$action$»/«$reaction$» stereotype then remove the active copy from the set of configuration copies.

$$Lc_{spec} \longleftarrow Lc_{spec} \setminus \{lc\},$$

   b. which is negation with «$forbidden$» stereotype then remove the active copy from the set of diagram copies representing forbidden behaviors and recording the realized scenario:

$$Lc_{forbidden} \longleftarrow Lc_{forbidden} \setminus \{lc\},$$

$$Forbidden \longleftarrow Forbidden \cup \{lc.sd\},$$

   c. which is assertion with «$expected$» stereotype then remove the active copy from the set of diagram copies representing expected behaviors and recording the realized scenario:

$$Lc_{expected} \longleftarrow Lc_{expected} \setminus \{lc\},$$

$$Expected \longleftarrow Expected \cup \{lc.sd\}.$$

Definition 6.2.3

The transition relations $c \xrightarrow{e}_{com} c'$ is defined recursively by the following rules:

1. $time\_context \longleftarrow \emptyset$,
2. $guards \longleftarrow \perp$,
3. For each active copy of the configuration *c* $\left(lc \in c.Lc_{spec} \cup c.Lc_{forbidden} \cup c.Lc_{expected}\right)$, for which there exists an event *e' strict-unifiable* with an event *e* violating ordering of events

$$\left(\exists_{e' \in lc.E_{violating}} : c.match_{s-unif}(e', e)\right):$$

   a. if $lc.mode \in \{action, check\}$ then remove the active copy form the set of configuration copies:

$$Lc_q \longleftarrow Lc_q \setminus \{lc\}, \text{ where: } q \text{ means } spec, forbidden \text{ or } expected.$$

   b. if $lc.mode = reaction$ then label the configuration as inconsistent final configuration:

$$c.type \longleftarrow violating.$$

4. For each sequence diagram $sd \in bSpec$, for which its set of first events contains an event e' *weak-unifiable* with the event e $\left(e' \in sd.E_{first} \wedge c.match_{w-unif}(e', e)\right)$, create active copy of the diagram and attach it to a respective set of active configurations:
   a. for $sd \in Sd_{spec}$:

$$Lc_{spec} \longleftarrow Lc_{spec} \cup \{lc = \langle sd, initial(sd), action\rangle\},$$

   b. for $sd \in Sd_{forbidden}$:

$$Lc_{forbidden} \longleftarrow Lc_{forbidden} \cup \{lc = \langle sd, initial(sd), check\rangle\},$$

   c. for $sd \in Sd_{expected}$:

$$Lc_{expected} \longleftarrow Lc_{expected} \cup \{lc = \langle sd, initial(sd), check\rangle\}.$$

5. For each active copy of the configuration c $\left(lc \in c.Lc_{spec} \cup c.Lc_{forbidden} \cup c.Lc_{expected}\right)$, for which the set of reachable events contains an event e' weak-unifiable with the event e $\left(e' \in lc.E_{reachable} \wedge c.match_{w-unif}(e', e)\right)$ do the following:

   a. for each synchronization event e'' at the diagram which proceeds the event e' and which has not been considered yet $\left(e'' \in lc.sd.E_{sync} \wedge e'' \preccurlyeq_{sd} e' \wedge e'' \notin\downarrow lc.s_{sd}\right)$, determine the transition:

$$c \xrightarrow{e''}_{sync} c',$$

   b. if the event e' is the first event within argument of an alternative combined fragment $(cf \in lc.sd.Cf \wedge cf.oper = alt)$, i.e.

$$\left(\exists_{cfa_q \in cf.CfA}: cfa_q \in allEnclosing(e') \wedge \nexists_{e'' \in lc.sd.E_{com}}: \left(cfa_q \in allEnclosing(e'') \wedge e'' \preccurlyeq_{sd} e'\right)\right)$$

   then exit the second argument of the combined fragment $\left(cfa_p \in cf.CfA \wedge cfa_p \neq cfa_q\right)$:

$$lc \leftarrow advance\left(lc, \langle cfa_p\ exit\rangle\right),$$

$$guards \longleftarrow guards \wedge cf.guard$$

   c. modify the active copy of the configuration by including the considered event:

$$lc \longleftarrow advance(lc, e'),$$

   d. determine unification of the events e and e':

$$unify(e', e), \qquad /* \text{ definition 6.2.4}$$

   e. attach time points to lifelines associated with realization of the event e':

$$time\_context \longleftarrow time\_context \cup e'.eventTime,$$

   f. if the event e' represents sending of a message and e'' is the event represents receiving of the message, then modify the lifelines associated with realization of the event e'':

$$lc \longleftarrow advance(lc, e''),$$

$$time\_context \longleftarrow time\_context \cup e''.eventTime,$$

g. for each enabled synchronization event e'' of the monitoring diagram $\left(lc \in c.Lc_{forbidden} \cup c.Lc_{expected} \wedge e'' \in lc.sd.E_{sync} \wedge e'' \in lc.E_{enabled}\right)$ determine the transition:

$$c \xrightarrow{e''}_{sync} c'.$$

Definition 6.2.4

The procedure of event unification is defined as follows:

$$\begin{array}{lll} unify(e', e) & \triangleq & unify(e'.msg, e.msg) \\ unify(msg', msg) & \triangleq & unify(msg'.arg, msg.arg), \\ unify(var', var) & \triangleq & 1.\ var'.\omega \leftarrow var.\omega, var'.\Phi \leftarrow var.\Phi, \\ & & 2.\ connect(var', var) \end{array} \quad /^* \text{ definition 6.2.5}$$

Definition 6.2.5

The binding variables procedure is defined as follows:

$$connect(x, y) \triangleq$$

1. $S \longleftarrow \{x, y\}$,
2. If the variable x or y were bounded already then merge the sets:

   a. $\exists_{S_x \in context}: x \in S_x \Longrightarrow \quad S \longleftarrow S \cup S_x$
   $context \longleftarrow context \setminus S_x$

   b. $\exists_{S_y \in context}: y \in S_y \Longrightarrow \quad S \longleftarrow S \cup S_y$
   $context \longleftarrow context \setminus S_y$

3. $context \longleftarrow context \cup S$.

7. Conclusions

UML sequence diagrams allow capturing requirements in a convenient way. Due to their simple, intuitive syntax and semantics, they are a suitable communication medium between analysts, developers, customers and end-users. Due to their focus on inter-object communication they are useful for specification of reactive systems, in particular real-time systems. Of course, they are not the only UML diagrams that may be used for real-time systems specification. For example, in (Roubtsova et al., 2000), system's specification consists of a class and object diagrams representing static, and state diagrams representing behavior aspect of the specification.

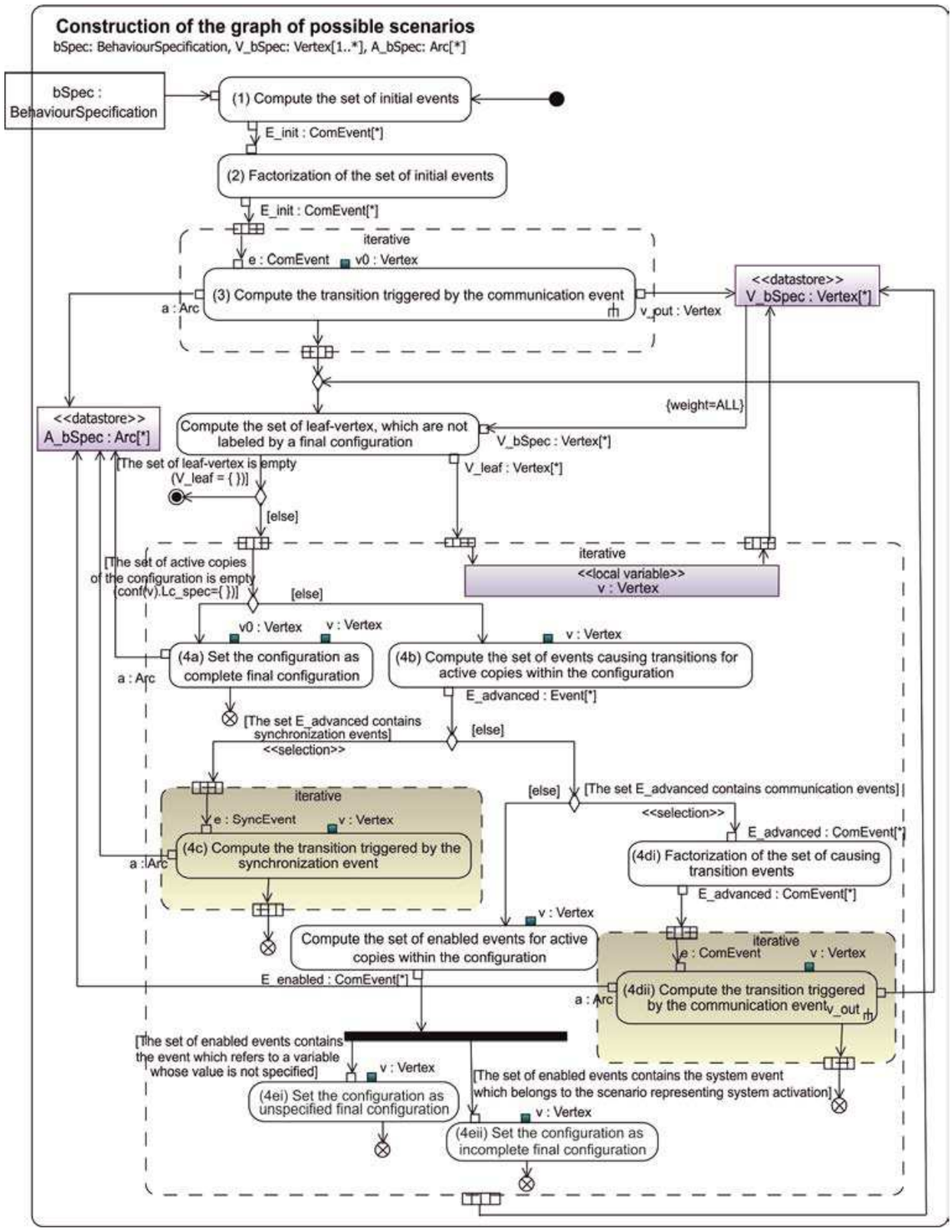

Fig. 20. The activity diagram presenting the construction of the graph of possible scenarios

In the chapter we present a way of effective application of UML sequence diagrams to real-time system specification. Assumption that specification of a system is represented by a set of sequence diagrams is close to the agile software development methodologies: each sequence diagram represents a single user story, and the set of sequence diagrams should represent complete description of the designed system.

In general, the proposed approach to a construction of system specification may be considered as a bottom-up approach – on the base of a set of user stories a complete specification and its semantics is derived. A consequence of the approach is the need to check consistency of the set of user stories. Another bottom-up approach basing on Live Sequence Charts is presented in (Harel & Marelly, 2003). The approach was also inspiration for our works. It is worth to note that application of Live Sequence Charts is further developed in (Maoz & Harel, 2011).

Opposite approaches to system specification are classified as top-down ones. Within these approaches, specification is constructed as a set of hierarchically ordered sequence diagrams. The diagrams at higher level of the hierarchy are composed from diagrams at lower level of the hierarchy, by means of some composition operators. An example of such approach, adopting Message Sequence Charts, is presented in (Cleaveland & Sengupta, 2006).

Lack of precise, formal semantics for UML sequence diagrams brings many interpretation problems. An illustration of this statement is more than a dozen proposals for the semantics of sequence diagrams, which are surveyed in (Micskei & Waeselynck, 2011). However, in contrast to our approach, it should be emphasized that these proposals concern only individual sequence diagrams, but not the set of diagrams.

The proposed approach to semantic definition of a set of UML sequence diagrams is based on transformation of the set into a graph of all possible scenarios. The graph enables system analyst to give answer to practical questions about consistency and completeness of scenarios, and thus about consistency and completeness of specification. Additionally, both consistency and definiteness may be checked automatically on the-fly. The checking algorithm was implemented as a programming tool supporting edition and analysis of specifications (Walkowiak, 2011). This tool has a form of plug-in to Visual Paradigm modeling tool.

8. References

Cleaveland, R. & Sengupta,B. (2006). Triggered Message Sequence Charts. *IEEE Transactions on Software Engineering* , Vol. 32, No. 8, (Aug. 2006), pp. (587 - 607), ISSN 0098-5589

Harel, D. & Marelly, R. (2003). Come, Let's Play: A Scenario-Based Approach to Programming, ISBN 3-540-00787-3, *Springer*

Harel, D. & Maoz, S. (2008). Assert and Negate Revisited: Modal Semantics for UML Sequence Diagrams, *Software and Systems Modeling,* Vol. 7, No. 2, (May 2008), pp. (237-252), ISSN 1619-1366

Huzar, Z. & Walkowiak, A. (2010). Specification of real-time systems using UML sequence diagrams, Przegląd Elektrotechniczny (Electrical Review), R. 86 NR 9/2010, pp. 226-229, ISSN 0033-2097

ITU-T (2004). Z. 120. Message Sequence Charts, available at http://www.itu.int/rec/T-REC-Z.120

Manna, Z. & Pnueli A. (1992). The Temporal Logic of Reactive and Concurrent Systems: Specification, ISBN 0387976647 , *Springer-Verlag*

Manna, Z. & Pnueli A. (1995). Temporal Verification of Reactive Systems: Safety, ISBN 0387944591, *Springer-Verlag*

Maoz, S. & Harel, D. (2011). On tracing reactive systems, Software and Systems Modeling, vol. 10, no. 4., pp. 447-468, ISSN 1619-1366

Micskei, Z. & Waeselynck, H. (2011). The many meanings of UML 2 Sequence diagrams: a survey, Software and Systems Modeling, vol. 10, no. 4., pp. 489-514, , ISSN 1619-1366

Nissanke, N. (1997). Realtime systems, ISBN 978-0136512745, *Prentice Hall*

OMG (2011). Unified Modeling Language, Superstructure, version 2.4.1. Available from: http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF/

Roubtsova, E.E., & van Katwijk, J., & Toetenel, W.J., & Pronk, C., & de Rooij, R.C.M. (2000). Specification of Real-Time Systems in UML, Electronic Notes in Theoretical Computer Science, vol. 39, no. 3.
URL: http//www.elsevier.nl/locate/entcs/volume39.html 13 pages

Störrle, H. (2004). Assert, Negate and Refinement in UML-2 Interactions, *Proceedings of 3rd Int. Workshop on Critical Systems Development with the UML (CSDUML'04)*, Lisbon, October 2004

Walkowiak, A. (2011). Specification of reactive systems, PhD Thesis (in Polish), Wrocław University of Technology, Faculty of Informatics and Management, Institute of Informatics

Real-Time Systems, Architecture, Scheduling, and Application

Edited by Dr. Seyed Morteza Babamir

ISBN 978-953-51-0510-7
Hard cover, 334 pages
Publisher InTech
Published online 11, April, 2012
Published in print edition April, 2012

This book is a rich text for introducing diverse aspects of real-time systems including architecture, specification and verification, scheduling and real world applications. It is useful for advanced graduate students and researchers in a wide range of disciplines impacted by embedded computing and software. Since the book covers the most recent advances in real-time systems and communications networks, it serves as a vehicle for technology transition within the real-time systems community of systems architects, designers, technologists, and system analysts. Real-time applications are used in daily operations, such as engine and break mechanisms in cars, traffic light and air-traffic control and heart beat and blood pressure monitoring. This book includes 15 chapters arranged in 4 sections, Architecture (chapters 1-4), Specification and Verification (chapters 5-6), Scheduling (chapters 7-9) and Real word applications (chapters 10-15).

How to reference

In order to correctly reference this scholarly work, feel free to copy and paste the following:

Zbigniew Huzar and Anita Walkowiak (2012). Specification and Validation of Real-Time Systems Using UML Sequence Diagrams, Real-Time Systems, Architecture, Scheduling, and Application, Dr. Seyed Morteza Babamir (Ed.), ISBN: 978-953-51-0510-7, InTech, Available from: http://www.intechopen.com/books/real-time-systems-architecture-scheduling-and-application/specification-and-validation-of-real-time-systems-using-uml-sequence-diagrams

INTECH
open science | open minds

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821

IntechOpen